L14 - Inheritance and Polymorphism

March 3, 2020

1 Inheritance and Polymorphism

We know we can inherit methods and fields from another class, and we can do that an arbitrary number of times.

[1]:
```
import java.util.ArrayList;

/**
 * This class stores information about a news feed post in a
 * social network. Posts can be stored and displayed. This class
 * serves as a superclass for more specific post types.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.2
 */
public class Postv2
{
    private String username;  // username of the post's author
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Constructor for objects of class Post.
     *
     * @param author    The username of the author of this post.
     */
    public Postv2(String author)
    {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<>();
    }

    /**
     * Record one more 'Like' indication from a user.
     */
```

```
    public void like()
    {
        likes++;
    }

    /**
     * Record that a user has withdrawn his/her 'Like' vote.
     */
    public void unlike()
    {
        if (likes > 0) {
            likes--;
        }
    }

    /**
     * Add a comment to this post.
     *
     * @param text  The new comment to add.
     */
    public void addComment(String text)
    {
        comments.add(text);
    }

    /**
     * Return the time of creation of this post.
     *
     * @return The post's creation time, as a system time value.
     */
    public long getTimeStamp()
    {
        return timestamp;
    }

    /**
     * Display the details of this post.
     *
     * (Currently: Print to the text terminal. This is simulating display
     * in a web browser for now.)
     */
    public void display()
    {
        System.out.println(username);
        System.out.print(timeString(timestamp));

        if(likes > 0) {
```

```
            System.out.println("  -  " + likes + " people like this.");
        }
        else {
            System.out.println();
        }

        if(comments.isEmpty()) {
            System.out.println("   No comments.");
        }
        else {
            System.out.println("   " + comments.size() + " comment(s). Click␣
↪here to view.");
        }
    }

    /**
     * Create a string describing a time point in the past in terms
     * relative to current time, such as "30 seconds ago" or "7 minutes ago".
     * Currently, only seconds and minutes are used for the string.
     *
     * @param time  The time value to convert (in system milliseconds)
     * @return      A relative time string for the given time
     */

    private String timeString(long time)
    {
        long current = System.currentTimeMillis();
        long pastMillis = current - time;      // time passed in milliseconds
        long seconds = pastMillis/1000;
        long minutes = seconds/60;
        if(minutes > 0) {
            return minutes + " minutes ago";
        }
        else {
            return seconds + " seconds ago";
        }
    }
}
```

[2]:
```
import java.util.ArrayList;

/**
 * This class stores information about a post in a social network news feed.
 * The main part of the post consists of a (possibly multi-line)
 * text message. Other data, such as author and time, are also stored.
 *
 * @author Michael Kölling and David J. Barnes
```

```
 * @version 0.2
 */
public class MessagePostv2 extends Postv2
{
    private String message;  // an arbitrarily long, multi-line message

    /**
     * Constructor for objects of class MessagePost.
     *
     * @param author    The username of the author of this post.
     * @param text      The text of this post.
     */
    public MessagePostv2(String author, String text)
    {
        super(author);
        message = text;
    }

    /**
     * Return the text of this post.
     *
     * @return The post's message text.
     */
    public String getText()
    {
        return message;
    }
}
```

[3]:
```
import java.util.ArrayList;

/**
 * This class stores information about a post in a social network news feed.
 * The main part of the post consists of a photo and a caption.
 * Other data, such as author and time, are also stored.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.2
 */
public class PhotoPostv2 extends Postv2
{
    private String filename;  // the name of the image file
    private String caption;   // a one line image caption

    /**
     * Constructor for objects of class PhotoPost.
     *
```

```
     * @param author    The username of the author of this post.
     * @param filename  The filename of the image in this post.
     * @param caption   A caption for the image.
     */
    public PhotoPostv2(String author, String filename, String caption)
    {
        super(author);
        this.filename = filename;
        this.caption = caption;
    }

    /**
     * Return the file name of the image in this post.
     *
     * @return The post's image file name.
     */
    public String getImageFile()
    {
        return filename;
    }

    /**
     * Return the caption of the image of this post.
     *
     * @return The image's caption.
     */
    public String getCaption()
    {
        return caption;
    }
}
```

[4]:
```
import java.util.ArrayList;

/**
 * The NewsFeed class stores news posts for the news feed in a
 * social network application.
 *
 * Display of the posts is currently simulated by printing the
 * details to the terminal. (Later, this should display in a browser.)
 *
 * This version does not save the data to disk, and it does not
 * provide any search or ordering functions.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.2
 */
```

```
public class NewsFeedv2
{
    private ArrayList<Postv2> posts;

    /**
     * Construct an empty news feed.
     */
    public NewsFeedv2()
    {
        posts = new ArrayList<>();
    }

    /**
     * Add a post to the news feed.
     *
     * @param post  The post to be added.
     */
    public void addPost(Postv2 post)
    {
        posts.add(post);
    }

    /**
     * Show the news feed. Currently: print the news feed details
     * to the terminal. (To do: replace this later with display
     * in web browser.)
     */
    public void show()
    {
        // display all posts
        for(Postv2 post : posts) {
            post.display();
            System.out.println();   // empty line between posts
        }
    }
}
```

1.0.1 Same demo as last time

[5]:
```
// create a new news feed
NewsFeedv2 nf2 = new NewsFeedv2();

PhotoPostv2 pp2 = new PhotoPostv2("Dave", "birb.jpg", "A birb");

nf2.addPost(pp2);
pp2.addComment("birb");
pp2.like();
```

```
pp2.like();
pp2.like(); // everyone loves birbs

MessagePostv2 mp2 = new MessagePostv2("Bill", "Nice birb");

nf2.addPost(mp2);
mp2.like();
mp2.like(); // nice wholesome post

nf2.show();
```

```
Dave
0 seconds ago  -  3 people like this.
   1 comment(s). Click here to view.

Bill
0 seconds ago  -  2 people like this.
   No comments.
```

This demo doesn't have a super helpful newsfeed - the `display()` method is from the `Post` superclass, not `MessagePost` or `PhotoPost`. This means that `display()` *has no idea about the unique fields and methods in the `MessagePost` and `PhotoPost` subclasses* - inheritance only works one way. That means that the superclass method can *only* work on the common fields in the superclass (likes, comments).

1.1 Static and Dynamic Type

The declared type of a variable is its *static type*. The type of the object a variable refers to is its *dynamic type*.

`Car c1 = new Car();`

`c1` has `Car` as both its static and dynamic type.

`Vehicle v1 = new Car();`

`v1` has a static type of `Vehicle` and a dynamic type of `Car`.

Note that the compiler will check for *static* type violations. Dynamic type violations are a *runtime* error.

1.2 Overriding

We can override a *superclass's* function definition by declaring a method *with the same signature* in the subclass.

Java will search for method names starting at the bottom of the inheritance hierarchy and then work its way up - and therefore find (and execute) the one in the lowest subclass first. This is called *overriding* - the subclass method is the *overriding* method and the superclass method is said to be *overridden*. The overridden method is not executed.

The overriding method has access to all the fields in whichever subclass it is in.

[23]:
```
public class Animal
{

    public Integer number_of_feet;

    public Animal()
    {
        number_of_feet = 0;
    }

    public Integer getFeet()
    {
        return number_of_feet;
    }

    public String speak()
    {
        return "Animals can't talk, silly :)";
    }
}

public class Cat extends Animal
{
    public Cat()
    {
        super();
        this.number_of_feet = 4;
    }
}

public class Bird extends Animal
{
    public Bird()
    {
        super();
        this.number_of_feet = 2;
    }

    public String speak()
    {
        return "I tawt I taw a puddy tat";
    }

    public String mute()
    {
```

```
        return super.speak();
    }
}
```

[37]:
```
Bird tweety = new Bird();

Cat sylvester = new Cat();

Animal roadrunner = new Bird();
```

[25]:
```
sylvester.speak();
```

[25]:
```
Animals can't talk, silly :)
```

[26]:
```
tweety.speak();
```

[26]:
```
I tawt I taw a puddy tat
```

[36]:
```
tweety.getFeet(); // this one comes from the superclass
```

[36]:
```
2
```

We can clearly see that the subclass version of `speak()` in `Bird` overrides the one in `Animal`. Since `Cat` doesn't have a `speak()` method, the one in `Animal` is executed. However, we can use `super.` to call the method from the superclass - like if you want a subclass to *extend* a method from the superclass.

This is known as **polymorphic method dispatch**. The variable can store objects of multiple types; but the actual method called depends on the object upon whom the method is called.

1.2.1 The `instanceof` operator

[49]:
```
roadrunner instanceof Animal
```

[49]:
```
true
```

[50]:
```
roadrunner instanceof Bird
```

[50]:
```
true
```

Identifies whether an object is of a particular type or of any subtype of it.

1.3 `Object` Class Methods

Methods in `Object` are inherited by all classes. Any of them may be overridden. For example we often override `Object.toString()` in order to give an object a useful string representation (for example, for debugging).

Calling `System.out.println()` will automatically call `toString()` since you can only print strings.

1.3.1 Overriding `equals()`

What does it mean for two of your objects to be the same?

- reference equality - literally the same object
- content equality - some or all content in the object is the same

Reference equality is dealt with by `==`. Content equality is dealt with by `equals()` methods - you can override `equals()` depending on what you want this to mean.

[ ]:
```
public boolean equals(Object obj)
{
    if ( this == obj ){
        return true; // literally the same
    }

    if (!(obj instanceof Animal))
    {
        return false; // not even the right type
    }

    // other class-specific comparisons here
}
```

[ ]:
```
public int hashCode()
// this method determines how things are store in HashMaps and HashSets
{
    return int 42;
}
```

1.4 `protected` access

`private` access is very restrictive and doesn't allow access of field and methods from subclasses; but `public` is awfully public. IN between, we can use `protected` which is accessible from subclasses only - not from other methods and classes.

[ ]:
```

```